Australian Government
Department of Defence
Defence Science and
Technology Organisation

Using Rotations to Build Aerospace Coordinate Systems

Don Koks

Electronic Warfare and Radar Division
Systems Sciences Laboratory

DSTO–TN–0640

ABSTRACT

Presented here are the main techniques necessary to understand rotations in three dimensions, for use with global visualisation and aerospace simulations. Relevant techniques can be extremely difficult to find in textbooks, so some useful examples are collected here to highlight these techniques.

The three standard aerospace coordinate systems are described and built using rotations. The mathematics of rotations is described, using both matrices and quaternions. The necessary calculations are given for analysing standard scenarios that involve the Global Positioning Satellite system for finding line-of-sight directions on Earth, as well as for visualising the world from a cockpit, and for converting to and from the standard software protocol for distributed interactive simulation environments.

Appendices then discuss combining rotations, conversions with a particular type of Euler angle convention, the dangers of confusing Euler angles with incremental rotations for software writers, and finally there is a short dicussion of interpolation of rotations in computing.

Published by

DSTO Systems Sciences Laboratory
P.O. Box 1500
Edinburgh, SA 5111
Australia

Telephone: (08) 8259 5555
Facsimile: (08) 8259 6567

© Commonwealth of Australia 2008
AR No. AR–013–424
August, 2008

Using Rotations to Build Aerospace Coordinate Systems

EXECUTIVE SUMMARY

This paper presents the main techniques necessary to understand three-dimensional rotations. The subject is not new, but can be very difficult to sort out and to explore in textbooks. Mathematics and physics texts that discuss the subject generally do so only in passing, and probably never cover any standard software protocols. The main literature on the subject seems to be that generated by the graphics computing community, since programmers need to use rotations when writing three-dimensional visualisation code. Nevertheless, code ease-of-use and ability to be passed on means that although there are many internet sites that talk about rotations, their information content is often biased towards whatever system or code their authors have become familiar with.

We begin by describing the three standard coordinate systems that are used for simulation of aerospace scenarios and the Global Positioning Satellite system, and how their defining axes can be built from rotations alone. We then describe the mathematics of rotations using both matrices and quaternions, defining Euler angles, and concentrating on the important matrix (or equivalently, quaternion) that allows any rotation about any axis to be made.

As examples of the techniques, we give the necessary calculations for dealing with standard scenarios involving the Global Positioning Satellite system, for finding line-of-sight directions on Earth, for visualising the world from a cockpit, and for converting to and from the standard software protocol for distributed interactive simulation environments.

Appendices then discuss combining rotations, conversions with a particular type of Euler angle convention, the dangers of confusing Euler angles with incremental rotations for software writers, and finally there is a short dicussion of interpolation of rotations in computing.

Author

Don Koks
Electronic Warfare and Radar Division

Don Koks completed a doctorate in mathematical physics at Adelaide University in 1996, with a dissertation describing the use of quantum statistical methods to analyse decoherence, entropy and thermal radiance in both the early universe and black hole theory. He holds a Bachelor of Science from the University of Auckland in pure and applied mathematics, and a Master of Science in physics from the same university with a thesis in applied accelerator physics (proton-induced X ray and γ ray emission for trace element analysis). He has worked on the accelerator mass spectrometry programme at the Australian National University in Canberra, as well as in commercial internet development.

Currently he is a Research Scientist with the Maritime Systems group in the Electronic Warfare and Radar Division at DSTO, specialising in jamming, three-dimensional rotations, and geolocation. He has published a book on mathematical physics called *Explorations in Mathematical Physics: the Concepts Behind an Elegant Language* (Springer, 2006).

Contents

Appendices

Glossary, Conventions, and Constants

$\boldsymbol{\alpha}$ L***a***titude of a point in the ECEF frame.

$\boldsymbol{\omega}$ L***o***ngitude of a point in the ECEF frame.

Aircraft's own axes x, y, z.

Aircraft's own axes as vectors in ECEF $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$.

Earth's elliptical cross-section This is *defined* by the following numbers:

$$\text{Semi-major: } a = 6{,}378{,}137 \text{ m}\,; \quad \text{semi-minor: } b = 6{,}356{,}752.3142 \text{ m.}$$

ECEF Earth-Centred, Earth-Fixed Frame. A rotating frame fixed to Earth; the global frame of this report within which all computations are done.

ECEF Cartesian Axes X, Y, Z.

Three Euler rotations act on a set of orthonormal vectors $\boldsymbol{x}_0, \boldsymbol{y}_0, \boldsymbol{z}_0$ to give, in order, $\boldsymbol{x}_1, \boldsymbol{y}_1, \boldsymbol{z}_1$, then $\boldsymbol{x}_2, \boldsymbol{y}_2, \boldsymbol{z}_2$, and finally $\boldsymbol{x}_3, \boldsymbol{y}_3, \boldsymbol{z}_3$.

GPS Global Positioning Satellite System.

Local Geographic Frame A frame based at our current position, usually on or near Earth's surface. Two common choices for its axes are:

NED North-East-Down set of axes.

ENU East-North-Up set of axes.

$\boldsymbol{n}$ A unit vector pointing along an axis of rotation. A column when used with matrices, and a row when used with quaternions. This might seem potentially confusing, but there's no ambiguity in practice; on the contrary, to continually write $\boldsymbol{n}$ and $\boldsymbol{n}^t$ (the transpose) just to emphasise a column or row would be tedious.

North, East, Down axes N, E, D.

North, East, Down axes as vectors in ECEF $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$.

Rotation matrix $R_{\boldsymbol{n}}(\theta)$ rotates by angle θ around unit vector $\boldsymbol{n}$ using the right-hand convention.

Rotation quaternion $Q_{\boldsymbol{n}}(\theta)$ rotates by angle θ around unit vector $\boldsymbol{n}$ using the right-hand convention.

$\boldsymbol{r}_{A\leftarrow B}$ Position of point A relative to point B; i.e., a vector pointing from B to A.

SLERP Spherical Linear Interpolation. Used for interpolating orientations using quaternions.

WGS-84 World Geodetic System 1984 standard for maps. This standard defines the oblate spheroid typically used to model Earth's shape, as used in this report.

1 Introduction

Simulating air scenarios often involves the ability to implement changes in orientation. The researcher might need to visualise changes in orientation, or describe the departure from flying "straight and level" for an aircraft flying at an arbitrary orientation, undergoing various heading changes, pitches, and rolls. Furthermore, these calculations are dependent on the aircraft's position on an ellipsoidal Earth, so that the modeller has a potential problem with various rotations that need to be performed.

Add to this the choice of different ways to rotate, and it's not surprising that rotations in three dimensions tend to be seen as overly difficult. To complicate the situation further, there are any number of internet graphics software sites that attempt to give information and explanations about rotations, but these are not always consistent in their notation, and are also not always trustworthy. There can be a gap between the explanations of those who understand the mathematics but who might not have any "on the ground" programming experience, and those who might not understand the mathematics but who do appreciate the practical or numerical problems involved with writing computer code that performs rotations. But wrongly interpreting the behaviour of rotated objects can lead to difficulties, as can be seen in a later section.

This paper attempts to explain some of the basics behind rotations. It is not intended to give lots of formulæ for doing all manner of rotations, since these are complicated enough to be unusable if the reader has no knowledge of just what is happening underneath. Instead, the plan is to make use of a minimum number of techniques in order to reinforce the main ideas. If some basic strategies applicable to solving common orientation scenarios can be outlined, then it's hoped that the reader can calculate some things from first principles. If a software package is used, or even just snippets of code, some knowledge of what is really happening goes a long way to helping the user validate or debug such code. Software documentation is not above using axes labels such as x, y, z and X, Y, Z, and then subsequently mixing up the correct cases; so a first-principles knowledge is sometimes necessary just to read that documentation.

2 Important Coordinate Systems

Understanding aerospace simulations usually assumes knowledge of three frames of reference:

The Earth-Centred, Earth-Fixed Frame is global, attached to Earth itself, and always uses both cartesian and polar coordinates.

The Local Geographic Frame is attached to Earth but based at where the aircraft currently is. It uses either of two different cartesian coordinate systems.

The Aircraft Frame is fixed to the aircraft and uses three cartesian axes, but tends to describe measurements by angles about these axes.

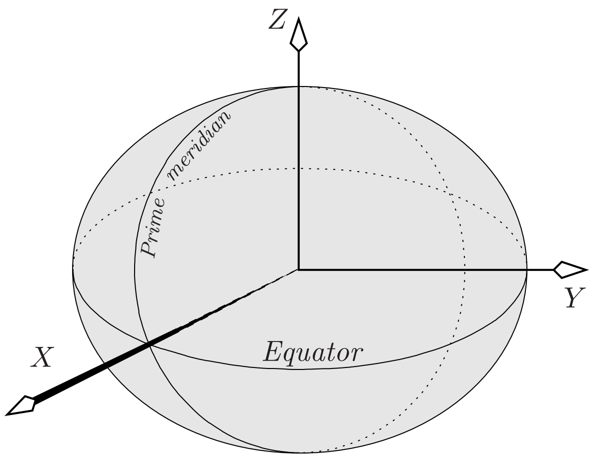

Figure 1: The Earth-Centred, Earth-Fixed frame can take a set of cartesian axes X, Y, Z with their origin at Earth's centre. The X-axis points to the 0° latitude, 0° longitude point. The Y-axis points to 0° latitude, 90° longitude. The Z-axis points to 90° latitude, along Earth's axis of rotation. Earth's surface is taken to be an oblate spheroid, i.e. having circular cross section at all latitudes, and a constant elliptical cross section through any meridian

There is plenty of scope here for difficulties and obscurity. In this section we describe each of these frames and coordinate choices in turn, before discussing how to go from one to another.

2.1 ECEF: the Earth-Centred, Earth-Fixed Frame

For most situations, and certainly all in this report, the Earth-Centred, Earth-Fixed Frame (ECEF) is a very useful stage on which to play out all scenarios, from the point of view of calculating orientations and directions for aircraft with global movements. All manipulations of numbers are done by using cartesian coordinate axes that are fixed in the ECEF.

The axes of the ECEF are right handed and have their origin at the centre of Earth, being fixed to its body: they rotate with it. They are quite adequate for describing situations on Earth, but are not so useful for describing satellite motion, such as scenarios that involve the GPS satellite system. The reason is because the ECEF frame is not quite inertial: because of its rotation, Newton's laws don't quite hold in it (unless we introduce complicated centrifugal and Coriolis forces), which means that satellite motion can be very complicated in the ECEF's coordinates. For such motion, a more encompassing frame tied to the fixed stars is used, but we won't need such a one in this report.

The ECEF has two common coordinate systems: a polar-type "latitude–longitude–height" called *geodetic coordinates*, and the simpler three cartesian axes X, Y, Z that are shown in Figure 1. Global positions are often given in lat–long–height coordinates, but since X, Y, Z are far easier to work with, we'll convert all lat–long–height to X, Y, Z. A shape for Earth commonly used in calculations is the oblate spheroid specified by the World

Geodetic System 1984 standard (WGS-84), which has a circular cross section at any given latitude, and whose cross section through any meridian is an ellipse, having identical axes lengths for all longitudes. These axes lengths are, by definition,

$$\begin{aligned} \text{Semi-major: } a &= 6{,}378{,}137 \text{ m} \\ \text{Semi-minor: } b &= 6{,}356{,}752.3142 \text{ m.} \end{aligned} \tag{2.1}$$

These numbers—especially b with its extra decimal places—might appear to be specifying a completely over-optimistic accuracy; but they are simply a combination of measurement and best fit to an ellipse, and so they just *define* WGS-84.

Any point has latitude α, longitude ω, and height h in this report. (There is no standard convention for latitude and longitude angles, so we have used the second letter of each word [“a” and “o”] and written them with Greek letters.) The latitude and longitude are shown in Figure 2, while the height of the point is the distance above the reference spheroid, i.e. along a line normal to it, *not* along a line extending from Earth’s centre.

The reason the latitude is defined with reference to the local normal and not Earth’s centre, is because by using the local normal, we can be sure that if two points on the same meridian have latitudes of say 10° and 50°, then the normal line (i.e. the local vertical) is guaranteed to rotate through 40° when passing from one to the other. So it’s very easy to compute how the vertical changes over Earth’s surface, and this would not be the case if latitude was measured relative to Earth’s centre.

Given lat-long-height values for any point, the corresponding X, Y, Z values are given in terms of a and b by:

$$\begin{aligned} X &= \left(\frac{a}{\sqrt{\cos^2\alpha + \frac{b^2}{a^2}\sin^2\alpha}} + h\right)\cos\alpha\ \cos\omega\,, \\ Y &= \left(\frac{a}{\sqrt{\cos^2\alpha + \frac{b^2}{a^2}\sin^2\alpha}} + h\right)\cos\alpha\ \sin\omega\,, \\ Z &= \left(\frac{b}{\sqrt{\frac{a^2}{b^2}\cos^2\alpha + \sin^2\alpha}} + h\right)\sin\alpha\,. \end{aligned} \tag{2.2}$$

The inverse transformation process is lengthier. Finding the longitude is the easiest part:

$$\cos\omega = \frac{X}{\sqrt{X^2+Y^2}}\,, \quad \sin\omega = \frac{Y}{\sqrt{X^2+Y^2}}\,, \tag{2.3}$$

so that for example in C and Matlab, ω can be found by using the single command `omega = atan2(Y,X)`. (Note that in Excel, the same function `atan2` has its arguments reversed, so would be `atan2(X,Y)` here.)

Latitude is more difficult, but can be calculated iteratively with reasonable ease. Begin with a first estimate

$$\alpha \simeq \tan^{-1}\left(\frac{a^2}{b^2}\frac{Z}{\sqrt{X^2+Y^2}}\right), \tag{2.4}$$

where e.g. Matlab's `atan` function will suffice (`atan2` is not necessary), and then iterate to refine this estimate using

$$\alpha = \tan^{-1}\left(\frac{a^2 \sin^2\alpha}{b^2 \sin\alpha\cos\alpha + \left(\sqrt{X^2+Y^2}\,\sin\alpha - Z\cos\alpha\right)\sqrt{a^2\cos^2\alpha + b^2\sin^2\alpha}}\right). \quad (2.5)$$

(This will fail if $Z = 0$, but then $\alpha = 0$ trivially anyway.) This expression for α converges very quickly; in fact about five decimal places of precision is obtained by only using the first estimate (2.4) and not even iterating at all. Finally, after an acceptable estimate of α has been obtained, the height is calculated:

$$h = \frac{\sqrt{X^2+Y^2}}{\cos\alpha} - \frac{a^2}{\sqrt{a^2\cos^2\alpha + b^2\sin^2\alpha}}\,. \quad (2.6)$$

2.2 The Local Geographic Frame

At any point on Earth's surface, the local geographic frame is defined by the almost flat ground and the vertical direction, and is relevant because it is the basic reference the aircraft flies against, defining straight and level flight and of course the direction of down. This is an intuitive frame for any scenario whose total extent is no more than some tens of kilometres.

Two coordinate systems are used: both cartesian with their origins at the point in question. In the North–East–Down or NED frame, shown in Figure 2, the local directions of north, east, and down define a right-handed set of three axes. An alternative choice of cartesian axes is the local East–North–Up set (ENU). In both axes sets, the north and east directions define a plane tangent to Earth's surface. At any point, north points along the local meridian, while east points along the local small circle of constant latitude.

The up/down or vertical direction is perpendicular to this plane. In general, the vertical direction does not intersect Earth's centre. Rather, it makes an angle of the latitude with the equatorial plane, shown as the angle α in Figure 2.

2.3 An Aircraft's Own Frame

The aircraft itself has its own frame described by a set of axes at rest relative to it, as shown in Figure 3. Conventionally the x-axis points forward along the nose, the y-axis points out along the starboard wing, and the z-axis points down. The x, y, z axes of an aircraft flying straight and level due north will match the local NED axes (i.e. x = north, y = east, z = down).

3 The Basic Tool: Vector Rotations

Switching between frames involves rotating vectors about specified axes, and just how to rotate vectors forms one of the core tools described in this report. In this section we

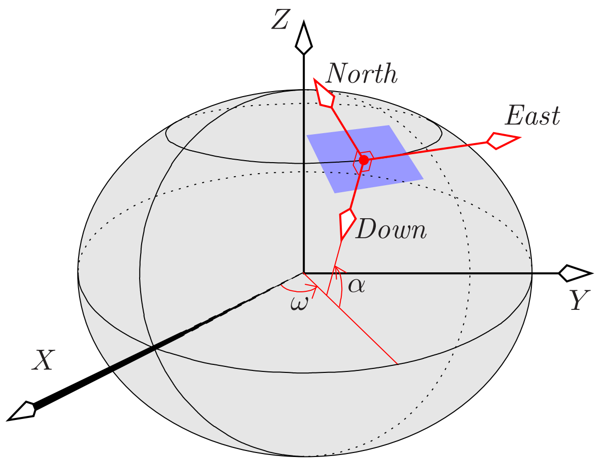

Figure 2: A plane tangent to Earth's hypothetical spheroidal surface contains the local East and North directions, which set those axes for the North–East–Down frame. The Down axis is normal to this plane. The latitude α of the point at the origin of the NED frame is set by the angle at which the Down axis intercepts Earth's equatorial plane. Because Earth is modelled as an oblate spheroid, the Down axis only points toward its centre when the NED origin is on the equator or at the poles. An alternative set of axes is East–North–Up

present the main formula for doing this. Surprisingly, this matrix formula for a general rotation in three dimensions can be hard to find in textbooks. Before giving the general expression, we'll start from more basic ideas.

3.1 Matrix Representation of an Orientation

How can we represent the orientation of a body in three dimensions? We can only do it in a relative way, by specifying how the body has been moved from some initial, or base, position. Suppose that the body is not translated, but its orientation is changed in some arbitrary way from this base position. Locate the body relative to the unmoving basis vectors, and imagine the body taking a copy of each basis vector along with it as it changes orientation. What are these new vectors? If the old ones are

$$\boldsymbol{i} \equiv \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{j} \equiv \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{k} \equiv \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \tag{3.1}$$

then suppose they are transformed into column vectors $\boldsymbol{i}', \boldsymbol{j}', \boldsymbol{k}'$. In that case, the change in orientation can be described by the matrix

$$A = \begin{bmatrix} \boldsymbol{i}' & \boldsymbol{j}' & \boldsymbol{k}' \end{bmatrix}. \tag{3.2}$$

in the sense that this multiplies $\boldsymbol{i}$ to give $\boldsymbol{i}'$, and similarly $\boldsymbol{j}, \boldsymbol{k}$. This expression is useful in that it gives the matrix that transforms an initial orientation to a final one immediately, without our needing to work out how the final orientation was produced.

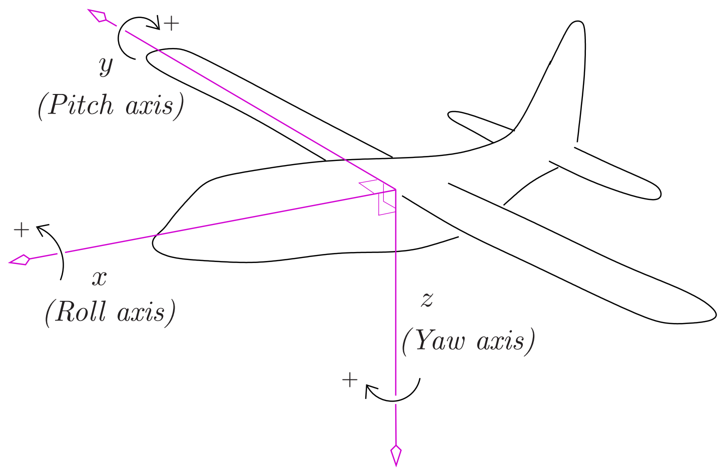

Figure 3: The x, y, z axes of the aircraft's own frame. These are axes about which the aircraft rolls (x), pitches (y), and yaws (z), where the direction of positive rotation for each is given by the right hand rule

If the body is rigid, then the linearity of matrix multiplication ensures that this matrix will describe the orientation of the whole body. That is, any other point in the base-positioned body that's described by a certain linear combination of the basis vectors $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$, will be described by the *same* linear combination of the new basis vectors $\boldsymbol{i}', \boldsymbol{j}', \boldsymbol{k}'$ in the new orientation. So the matrix A multiplies any vector to produce its orientated form.

The same result holds for the more trivial two dimensional case, where we rotate all vectors through θ in the xy plane, positively about the z-axis. The basis vectors then map as:

$$\boldsymbol{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \longrightarrow \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = \boldsymbol{i}', \quad \boldsymbol{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} = \boldsymbol{j}', \tag{3.3}$$

giving the familiar rotation matrix

$$A = \begin{bmatrix} \boldsymbol{i}' & \boldsymbol{j}' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{3.4}$$

3.2 Euler Angles

In two dimensions the above transformation cannot help but be a rotation, but in three dimensions it's not explicitly so; it simply produces a final orientation given some initial orientation. Nevertheless, it can be shown to be producible by just one rotation around some axis, where that axis needn't be any of the x, y, or z axes (and in general won't be). This is known as *Euler's theorem*.

The literature actually spends more time on the fact that the final orientation can be produced by successive rotations around the space-fixed x, y, z axes, through what are known as *Euler angles*. There are many different conventions for specifying Euler angles, but in any of them, three angles are required to describe a general change in orientation. Some authors leave one axis out and repeat another, so that the orientations are performed around, for example, the x-axis, then y-axis, then x-axis again (and all similar choices of three distinct rotations will also work, such as z, y, z etc.—but not e.g. z, y, y, since this is really only two distinct rotations). Some rotate around the x-axis, then the *new* y-axis (better called y'), then the new x-axis (now called x'), although this can be shown to reduce to a similar expression involving the same angles but in a different order, around the space-fixed x, y, x axes. The possibilities are many and confusing.

Two Euler angles are also used in the field of aircraft tracking, where for example a radar might be used to follow the motion of an aircraft. At any moment, the aircraft's position can be specified by azimuthal and elevation angles, as if we were using a telescope with a standard tripod to sight the aircraft. When the aircraft is nearly overhead, the telescope becomes difficult to move easily, because the closer we come to pointing it at the zenith, the more its azimuthal motion (motion about the vertical axis) constricts its tube to making smaller movements. This means that small movements around the zenith translate to large changes in the azimuthal angle, which can be difficult to achieve evenly. Mechanically this constricted movement is called *gimbal lock*. In using these two angles to track an aircraft, rapid changes in azimuth can be difficult to model in any numerical algorithm, and the term gimbal lock has also come to be applied to numerical difficulties. The result is that it has given Euler angles a bad name. But the problem is purely one of numerical stability; there's nothing wrong with using the two Euler angles in principle.

This constriction of motion about one axis also causes mechanical problems with some gyroscopes. To picture why this might be, imagine a very simple (and not very useful) example of an inertial navigation system: a horizontal spinning flywheel carried on a yoke attached to hinges at the end of each wingtip, and held underneath the aircraft while it flies straight and level. The flywheel's axis determines the up/down direction used by the autopilot to help keep the aircraft on course. The aircraft can pitch up and down and yaw in the local north–east plane without affecting the flywheel's motion. But if the human pilot tries to roll, the flywheel will be forced out of its spin plane. In such an event, the inertial navigation system will be fooled into believing that the aircraft has begun to develop some serious instabilities which it will try to correct, with perhaps disastrous consequences. While this is a big simplification of a real system, it does encapsulate the problem that can happen with a real gyroscope when the plane's attitude reaches some extreme position.

One way of avoiding numerical instabilities in tracking algorithms is to use Euler's theorem, which does away with having to use three angles around the three fixed axes. So it is that our matrix A in (3.2) describes this rotation. In principle it should be sufficient for all purposes, although in practice its nine entries can require a powerful ability for number crunching, especially when used to describe motions where the orientation angles are changing with time.

3.3 The Workhorse: Rotating in Three Dimensions

The matrix A in (3.2) describes the rotation that changes a body's orientation, provided we know how the basis vectors change. But usually we are not given the final position of those vectors. In general, we need to be able to rotate an arbitrary vector around an arbitrary axis, and this is a slightly different task to be done.

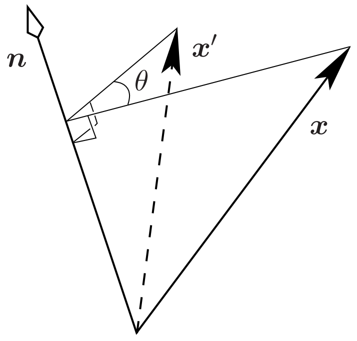

Figure 4: The matrix $R_{\boldsymbol{n}}(\theta)$ rotates $\boldsymbol{x}$ in a right-handed sense around the unit vector $\boldsymbol{n}$ by an angle θ to produce $\boldsymbol{x}'$. That is, $\boldsymbol{x}' = R_{\boldsymbol{n}}(\theta)\,\boldsymbol{x}$

This rotation can be done by multiplying the vector by a matrix $R_{\boldsymbol{n}}(\theta)$ that we write down in this section. $R_{\boldsymbol{n}}(\theta)$ specifies a right-handed rotation through some angle θ about an axis aligned with a unit vector $\boldsymbol{n}$. (Requiring $\boldsymbol{n}$ to have unit length makes the expressions simpler.) It rotates the vector $\boldsymbol{x}$ to give $\boldsymbol{x}'$, as shown in Figure 4:

$$\boldsymbol{x}' = R_{\boldsymbol{n}}(\theta)\,\boldsymbol{x}\,. \tag{3.5}$$

Write the unit vector that points along the axis of rotation as

$$\boldsymbol{n} = \begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix}, \tag{3.6}$$

where $\boldsymbol{n}$ can point in either direction along the axis, but changing the choice of that direction will change the sign required for the rotation angle θ, since the rotation matrix $R_{\boldsymbol{n}}(\theta)$ obeys the right hand rule for rotation. The rotation matrix can be written in the following way, which is the main equation of this report:

$$\begin{aligned} R_{\boldsymbol{n}}(\theta) &= (1-\cos\theta)\begin{bmatrix} n_1^2 & n_1 n_2 & n_1 n_3 \\ n_2 n_1 & n_2^2 & n_2 n_3 \\ n_3 n_1 & n_3 n_2 & n_3^2 \end{bmatrix} + \cos\theta\, I_3 + \sin\theta \begin{bmatrix} 0 & -n_3 & n_2 \\ n_3 & 0 & -n_1 \\ -n_2 & n_1 & 0 \end{bmatrix} \\ &= (1-\cos\theta)\,\boldsymbol{n}\boldsymbol{n}^t + \cos\theta\, I_3 + \sin\theta\, n^{\times}\,, \end{aligned} \tag{3.7}$$

where $\boldsymbol{n}^t$ is the transpose of $\boldsymbol{n}$, I_3 is the 3×3 identity matrix, and

$$n^{\times} \equiv \begin{bmatrix} 0 & -n_3 & n_2 \\ n_3 & 0 & -n_1 \\ -n_2 & n_1 & 0 \end{bmatrix}, \tag{3.8}$$

so-named because

$$n^{\times}\boldsymbol{x} = \boldsymbol{n}\times\boldsymbol{x}\,, \tag{3.9}$$

(with the n written nonbold to emphasise its matrix form). The rotation matrix $R_{\boldsymbol{n}}(\theta)$ is called *orthogonal*, by which is meant $RR^t = R^t R = 1$, and as well its determinant is always 1. Equations (3.5) and (3.7) will be used repeatedly in this report to break more complicated procedures up into single rotations that are easy to visualise and easy to code. They are, in fact, the central equations of the whole of rotation theory.

Example of the use of (3.7): Rotate the vector $(2,0,0)$ by 90° about the y-axis. What vector results? The required rotation matrix is $R_y(90°)$ (where by the subscript y is meant $\boldsymbol{n} = (0,1,0)^t$). Equation (3.7) gives it as

$$R_y(90°) = \boldsymbol{n}\boldsymbol{n}^t + n^\times = \begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}0 & 1 & 0\end{bmatrix} + \begin{bmatrix}0 & 0 & 1\\0 & 0 & 0\\-1 & 0 & 0\end{bmatrix} = \begin{bmatrix}0 & 0 & 1\\0 & 1 & 0\\-1 & 0 & 0\end{bmatrix}. \tag{3.10}$$

(Actually, in this simple example we can calculate $R_y(90°)$ alternatively using (3.2). Simply note that the basis vectors rotate as

$$\begin{bmatrix}1\\0\\0\end{bmatrix} \longrightarrow \begin{bmatrix}0\\0\\-1\end{bmatrix}, \qquad \begin{bmatrix}0\\1\\0\end{bmatrix} \longrightarrow \begin{bmatrix}0\\1\\0\end{bmatrix}, \qquad \begin{bmatrix}0\\0\\1\end{bmatrix} \longrightarrow \begin{bmatrix}1\\0\\0\end{bmatrix}, \tag{3.11}$$

so that (3.2) yields $R_y(90°)$ trivially.) The required rotated vector is then

$$R_y(90°)\begin{bmatrix}2\\0\\0\end{bmatrix} = \begin{bmatrix}0 & 0 & 1\\0 & 1 & 0\\-1 & 0 & 0\end{bmatrix}\begin{bmatrix}2\\0\\0\end{bmatrix} = \begin{bmatrix}0\\0\\-2\end{bmatrix}, \tag{3.12}$$

as expected.

An example of combining two rotations is given in Appendix A. As stated by Euler's theorem, the effect of these is just one rotation, although the values of the resulting equivalent axis and angle turned through are not obvious at all from the original two axes and angles.

The rows of a rotation matrix will gradually lose their mutual orthonormality because of rounding errors under repeated iterations in a computer programme. This wandering can be kept in check by periodically re-orthogonalising, accomplished by

$$R \to R\left(R^t R\right)^{-1/2}. \tag{3.13}$$

Rotation Order and its Possible Blind Application

Since matrix multiplication is not commutative, two rotations are also not in general commutative: the result depends on the order in which they are done. However, infinitesimal rotations *are* commutative. This fact can be used to our advantage when writing computer code that takes inputs from a joystick, in order to rotate a scene (say, in a flight simulator). The pilot might induce both a pitch and a roll, but only by applying each of these in tiny increments will the software successfully reproduce the effect that the pilot wants.

Nevertheless, it's possible to go to an extreme of demanding a set rotation order that does *not* mimic what the user of a software package wants, but that does do something highly misleading. In fact, a search through the internet shows that this mistake seems to be a classic of computer graphics programming, and has caused some in that community to distrust the mathematics of rotation. An example of this incorrect application of rotations is discussed in Appendix C.

3.4 Rotating using Quaternions

Although the rotation matrix $R_{\boldsymbol{n}}(\theta)$ has nine entries, it's really only built from three numbers: the angle θ turned through, and any two components of $\boldsymbol{n}$; the third component of $\boldsymbol{n}$ is then implied, since $\boldsymbol{n}$ has unit length. Using $R_{\boldsymbol{n}}(\theta)$ can be computationally inefficient, since not only do all nine components need to be manipulated, but if the matrix is applied within a loop, perhaps thousands of times, numerical inaccuracies can degrade its orthogonality. That is, its rows (or equivalently columns) will slowly lose their mutual orthonormality. It can certainly be periodically re-orthogonalised after an appropriate number of iterations, as shown in (3.13). Nevertheless, the question arises as to whether any way exists of rendering the matrix down to its basic three numbers, and perhaps eliminating some of the numerical complexity in the process.

We are always free to construct any quantity from those three numbers. For instance we could specify a rotation through θ around (n_1, n_2, n_3) by a new object written as (θ, n_1, n_2). This is very concise, but the catch is that we would also have to define how this object acts on a vector to rotate it. The result is not anything simple, so we gain nothing by doing this.

But all is not lost. If we allow a little more complexity, and construct a new object $Q_{\boldsymbol{n}}(\theta)$ from θ, n_1, n_2, n_3 along with a sine and cosine, then this turns out to be simple enough to offset the more complicated way it has to interact with vectors. Here $\boldsymbol{n}$ is taken to be the unit *row* vector (n_1, n_2, n_3):

$$Q_{\boldsymbol{n}}(\theta) \equiv \left(\cos\frac{\theta}{2}, \boldsymbol{n}\sin\frac{\theta}{2}\right). \tag{3.14}$$

This new object $Q_{\boldsymbol{n}}(\theta)$ is called a *rotation quaternion.* More general quaternions (not for rotating) are composed of a number and a vector: $(a_0, \boldsymbol{a})$. Two quaternions multiply in the following way:

$$(a_0, \boldsymbol{a})\,(b_0, \boldsymbol{b}) \equiv (a_0 b_0 - \boldsymbol{a}\cdot\boldsymbol{b},\ \ a_0\boldsymbol{b} + b_0\boldsymbol{a} + \boldsymbol{a}\times\boldsymbol{b})\,, \tag{3.15}$$

so that the squared length of a quaternion is defined to be (in analogy to the length of a vector):

$$\left|(a_0, \boldsymbol{a})\right|^2 \equiv a_0^2 + |\boldsymbol{a}|^2\,. \tag{3.16}$$

We can see straight away that a rotation quaternion $Q_{\boldsymbol{n}}(\theta)$ has unit length, which is the equivalent property of rotation quaternions as orthogonality and a unit determinant is of rotation matrices.

Using rotation quaternions, a *row* vector $\boldsymbol{x}$ can be rotated through an angle θ around a unit vector $\boldsymbol{n}$ by applying a double multiplication:

$$(0, \boldsymbol{x}') = Q_{\boldsymbol{n}}(\theta)\ (0, \boldsymbol{x})\, Q_{-\boldsymbol{n}}(\theta)\,, \tag{3.17}$$

where quaternion multiplication is associative, so either product can be done first. As an example, in (3.12) we used a matrix to rotate $(2, 0, 0)$ by 90° about the y-axis to give $(0, 0, -2)$. Contrast the matrix approach with the quaternion approach:

$$(0, \boldsymbol{x}') \quad = \quad \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}(0,1,0)\right)\ (0,2,0,0)\ \left(\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}(0,1,0)\right)$$

$$= (1,0,1,0)\,(0,1,0,0)\,(1,0,-1,0)$$
$$= (0,0,0,-2)\,, \tag{3.18}$$

so that the result is $(0,0,-2)$ as expected.

Combining two rotations using quaternions is straightforward. Just as a matrix rotation R_1 followed by R_2 is equivalent to a single matrix rotation $R_2\,R_1$, a quaternion rotation Q_1 followed by Q_2 is equivalent to a single quaternion rotation $Q_2\,Q_1$.

Finally, just as a rotation matrix will slowly lose orthogonality under repeated iterations in a software routine, so a rotation quaternion's length will slowly wander from one in the same circumstances. It can be periodically reset by dividing each of the quaternion's four elements by the length calculated from (3.16), which is a very much simpler task than re-orthogonalising a matrix in (3.13).

Matrix Cost versus Quaternion Cost

How expensive is using matrices as opposed to using quaternions? While (3.18) might look succinct, it does hide some effort, especially in the two cross products that have been done. In Table 1 we give simple counts of operations required to rotate a vector, if done with pen and paper. No allowance for numerical optimisation is made since that forms a field in itself. It can be seen that quaternions are easier to build, while matrices are easier to use; thus, depending on the application, it can be useful to convert between the two subject to how much of each process needs to be done.

Table 1: First-principles computation costs of matrices and quaternions, for the build and one use of each. The numbers shown are not optimised

	Matrix	Quaternion
Build using (3.7) and (3.14):	2 trigonometric functions, 10 additions, 21 multiplications.	2 trigonometric functions, no additions, 4 multiplications.
One rotation using (3.5) and (3.15, 3.17):	6 additions, 9 multiplications.	24 additions, 30 multiplications.
Multiplication	27 multiplications, 18 additions.	16 multiplications, 12 additions.

The benefit of using quaternions for iterative calculations is that the resetting of a quaternion's length to 1 requires a simple division by its length, calculated from (3.16). On the other hand, re-orthogonalising a matrix from (3.13) is a much more complicated affair that involves the lengthy procedure of matrix diagonalisation. Although there is room for optimisation in this process, it cannot compete with quaternion normalisation for simplicity. How often such a resetting or re-orthogonalising needs to be done is presumably a function of the numerical complexity of the problem being tackled.

In fact, it might be thought that rather than use the expensive (3.13) to re-orthogonalise the matrix, that a better approach would be to convert the matrix to a quaternion, normalise that, and then convert back to a matrix. This can be done using the analysis of Appendix A, by applying (A1) to produce the angle and axis from the matrix, that are then used to build the quaternion via (3.14), which is then normalised and converted back to a matrix using (3.7). But whether this is really viable might depend on the numerical accuracy of the software. As it is, (A1) only samples the matrix's three diagonal elements to calculate the angle θ, perhaps producing too inaccurate a result to be used in building the quaternion. The quaternion could be produced by using all of the matrix elements; but it's not clear what work has been done in this area for doing this efficiently and accurately.

4 Rotating Axes to Change Coordinates

The building block that allows all manner of aerospace re-orientations to be done, for various pitches, rolls, yaws, changes of latitude/longitude, and local geographic frames, is the act of rotating each of the vectors that represent the axes of some frame.

There are two basic tasks to consider in this type of aerospace calculation. First, given a place on Earth (usually in lat–long–height coordinates), we need to construct the local geographic frame's axes, such as NED. Second, an aircraft can be introduced into this NED frame, and its orientation found relative to that frame.

4.1 Constructing NED Axes for the Local Geographic Frame

Given the lat–long–height of a place on or near Earth, the first thing we wish to do is find the local NED axes. The technique used is the primary one of this report. We build an initial set of NED axes in a place where it's simple to determine them, and then we rotate these around to the required place.

The simplest place to construct an initial set of NED axes is on the junction of the Equator and the prime meridian, since this has a latitude and longitude of $\alpha = \omega = 0°$. Represent each axis by a unit vector in the ECEF frame, which *all* calculations are being done within:

$$\boldsymbol{N}_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} , \quad \boldsymbol{E}_0 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} , \quad \boldsymbol{D}_0 = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} . \tag{4.1}$$

The scenario is shown in Figure 5. Now rotate each of the $\boldsymbol{N}_0, \boldsymbol{E}_0, \boldsymbol{D}_0$ vectors in two steps. The first rotation is about the $\boldsymbol{N}_0$ vector by the longitude ω, taking care to realise that the rotation matrix will obey the right hand rule. This rotation creates an intermediate triplet of vectors, taking $\boldsymbol{N}_0 \to \boldsymbol{N}_1$, $\boldsymbol{E}_0 \to \boldsymbol{E}_1$, $\boldsymbol{D}_0 \to \boldsymbol{D}_1$. (Of course, $\boldsymbol{N}_1 = \boldsymbol{N}_0$, but for clarity we keep each triplet together notationally.)

Now rotate each of the intermediate set by the latitude α about $-\boldsymbol{E}_1$; again we've been careful to note the use of the right hand rule, so need to specify the axis of rotation as $-\boldsymbol{E}_1$. (We can certainly use $+\boldsymbol{E}_1$, but will have to rotate by $-\alpha$ about this. The rotation matrix or quaternion will be unchanged.) Because the original $\boldsymbol{N}_0, \boldsymbol{E}_0, \boldsymbol{D}_0$ vectors each have

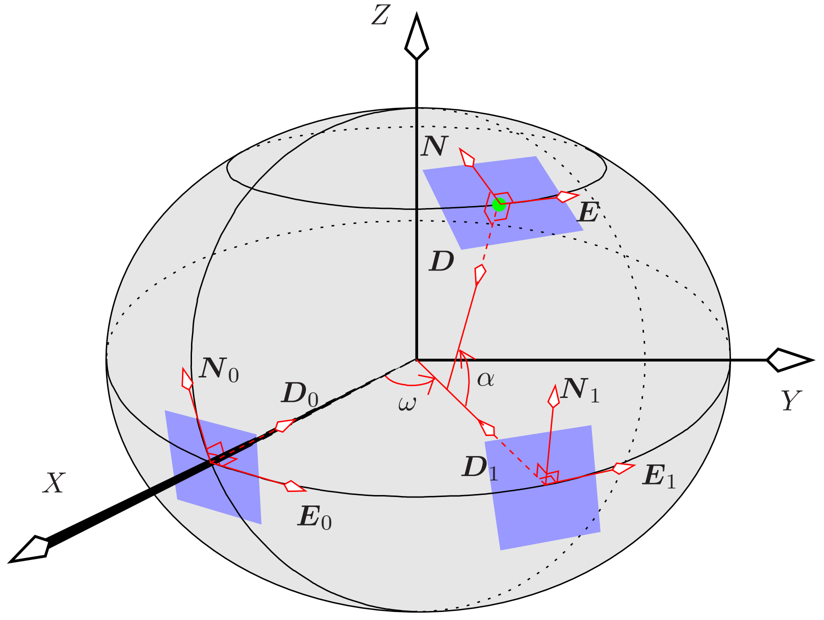

Figure 5: Constructing local north–east–down axes $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$ at a given point of known latitude and longitude. Start with the three vectors $\boldsymbol{N}_0, \boldsymbol{E}_0, \boldsymbol{D}_0$. Construct the intermediate set $\boldsymbol{N}_1, \boldsymbol{E}_1, \boldsymbol{D}_1$ by rotating the original set through the longitude ω about $\boldsymbol{N}_0$. Then rotate $\boldsymbol{N}_1, \boldsymbol{E}_1, \boldsymbol{D}_1$ about $-\boldsymbol{E}_1$ by the latitude α, according to the right hand rule, to give the final set $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$. Or equivalently, rotate the intermediate set about $\boldsymbol{E}_1$ through minus *the latitude. The resulting vectors are the local north–east–down axes*

unit length, they automatically satisfy the requirement for the axis vector $\boldsymbol{n}$ to be of unit length in (3.7). Notice that the clever way that latitude has been defined means we needn't worry that Earth's shape is not spherical; two places separated by say 50° latitude on the same meridian will certainly have a 50° angle between their North vectors, and a 50° angle between their Down vectors.

The sequence of steps is:

$$\begin{aligned}
\boldsymbol{N}_1 &= R_{\boldsymbol{N}_0}(\omega)\,\boldsymbol{N}_0 \qquad = \boldsymbol{N}_0\\
\boldsymbol{E}_1 &= R_{\boldsymbol{N}_0}(\omega)\,\boldsymbol{E}_0\\
\boldsymbol{D}_1 &= R_{\boldsymbol{N}_0}(\omega)\,\boldsymbol{D}_0\\[1em]
\boldsymbol{N} &= R_{-\boldsymbol{E}_1}(\alpha)\,\boldsymbol{N}_1\\
\boldsymbol{E} &= R_{-\boldsymbol{E}_1}(\alpha)\,\boldsymbol{E}_1 \qquad = \boldsymbol{E}_1\\
\boldsymbol{D} &= R_{-\boldsymbol{E}_1}(\alpha)\,\boldsymbol{D}_1\,.
\end{aligned} \tag{4.2}$$

The resulting vectors $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$ are the local north–east–down axes, and can be used for further calculations. In the interest of pedagogy, we have made the steps as alike as possible, but in practice they can be heavily reduced to

$$\boldsymbol{E} = R_{\boldsymbol{N}_0}(\omega)\,\boldsymbol{E}_0$$

$$\begin{aligned} \boldsymbol{N} &= R_{-\boldsymbol{E}}(\alpha)\,\boldsymbol{N}_0 \\ \boldsymbol{D} &= \boldsymbol{N} \times \boldsymbol{E}\,. \end{aligned} \tag{4.3}$$

This reduction is purely a question of pedagogy versus computational speed. Equations (4.2) allow for a step by step follow-through of the process, which is useful for writing transparent computer code or debugging in more complicated situations where more than two rotations need to be performed. In this two-rotation case, it's certainly not difficult to use (4.3) without having to think along the lines of (4.2) at all. But situations requiring more than two rotations will not be so easily abbreviated.

The following example draws many of these ideas together.

Example: *If we are in Adelaide and Earth is transparent, what is the compass bearing of Brussels if we are looking straight at it through Earth?*

We'll calculate ECEF position vectors of both Adelaide and Brussels, subtracting one from the other to find the position vector of Brussels relative to Adelaide, and then express this in the local NED axes at Adelaide using dot products. It's helpful to write the vectors in the following way, where A = Adelaide, B = Brussels, and C = some other useful point. The vector giving the position of B relative to A can be written $\boldsymbol{r}_{B\leftarrow A}$, and the following are all true in general, as well as being useful for writing down vector relationships without needing to draw pictures:

$$\begin{aligned} \boldsymbol{r}_{B\leftarrow A} &= -\boldsymbol{r}_{A\leftarrow B} \\ \boldsymbol{r}_{B\leftarrow A} &= \boldsymbol{r}_{B\leftarrow C} - \boldsymbol{r}_{A\leftarrow C} \\ \boldsymbol{r}_{B\leftarrow A} &= \boldsymbol{r}_{B\leftarrow C} + \boldsymbol{r}_{C\leftarrow A} \end{aligned} \tag{4.4}$$

The cities' positions are:

Adelaide:	latitude $\alpha = -34.9°$,	longitude $\omega = 138.5°$,
Brussels:	latitude $\alpha = 50.8°$,	longitude $\omega = 4.3°$.

Use (2.2) to find the position of each city relative to Earth's centre (which is exactly what the ECEF coordinates specify): set C = Earth's centre.

$$\boldsymbol{r}_{A\leftarrow C} = \begin{bmatrix} -3.92 \\ 3.47 \\ -3.63 \end{bmatrix} \times 10^6 \text{ metres}, \quad \boldsymbol{r}_{B\leftarrow C} = \begin{bmatrix} 4.03 \\ 0.30 \\ 4.92 \end{bmatrix} \times 10^6 \text{ metres}. \tag{4.5}$$

The position of Brussels relative to Adelaide is then

$$\boldsymbol{r}_{B\leftarrow A} = \boldsymbol{r}_{B\leftarrow C} - \boldsymbol{r}_{A\leftarrow C} = \begin{bmatrix} 7.95 \\ -3.17 \\ 8.55 \end{bmatrix} \times 10^6 \text{ metres}. \tag{4.6}$$

This vector is still an ECEF vector, although we can consider it to start at Adelaide and point to Brussels. To calculate the bearing of Brussels as seen from Adelaide, we need to get the local north and east components of $\boldsymbol{r}_{B\leftarrow A}$. To do this, we'll first calculate

Adelaide's local north, east, and down axis vectors, labelled $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$ in Figure 5, all in ECEF coordinates, using (4.2). For this we need $R_{\boldsymbol{N}_0}(\omega)$, constructed from (3.7):

$$\boldsymbol{N}_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \omega = 138.5^\circ, \quad R_{\boldsymbol{N}_0}(\omega) \simeq \begin{bmatrix} -0.74896 & -0.66262 & 0 \\ 0.66262 & -0.74896 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.7}$$

This $R_{\boldsymbol{N}_0}(\omega)$ then acts in (4.2) to produce $\boldsymbol{N}_1, \boldsymbol{E}_1, \boldsymbol{D}_1$. We then use $-\boldsymbol{E}_1$ to construct the next rotation matrix $R_{-\boldsymbol{E}_1}(\alpha)$, which finally produces the $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$. The steps are easy to programme and are omitted; the resulting axes are

$$\boldsymbol{N} = \begin{bmatrix} -0.429 \\ 0.379 \\ 0.820 \end{bmatrix}, \quad \boldsymbol{E} = \begin{bmatrix} -0.663 \\ -0.749 \\ 0 \end{bmatrix}, \quad \boldsymbol{D} = \begin{bmatrix} 0.614 \\ -0.543 \\ 0.572 \end{bmatrix}. \tag{4.8}$$

The bearing of Brussels is calculated by projecting $\boldsymbol{r}_{B \leftarrow A}$ into the local north–east plane. The projection can be visualised as an arrow drawn on the ground, so that its angle from north is the required bearing. The projection in the local north–east plane is simply given by the components of $\boldsymbol{r}_{B \leftarrow A}$ on the local north and east axes. These are dot products:

$$\begin{aligned} \text{North component} &= \boldsymbol{r}_{B \leftarrow A} \cdot \boldsymbol{N} = 2.4035 \times 10^6 \text{ metres,} \\ \text{East component} &= \boldsymbol{r}_{B \leftarrow A} \cdot \boldsymbol{E} = -2.8958 \times 10^6 \text{ metres.} \end{aligned} \tag{4.9}$$

(Note that these equations assume the axis vectors $\boldsymbol{N}, \boldsymbol{E}$ have unit length; another reason to prefer unit length axis vectors from the start.) The units here are immaterial; the only thing that matters is the ratio of the components, so that Brussels' bearing is

$$360^\circ - \tan^{-1} \frac{2.8958}{2.4035} \simeq 310^\circ, \tag{4.10}$$

or roughly north-west. Apart from the initial change of variables in (2.2), this whole calculation has been accomplished using just rotations and dot products. We could of course have streamlined it using (4.3) as well as omitting the calculation of the local Down vector $\boldsymbol{D}$, but the point here is pedagogy, not efficiency.

4.2 The World as Seen by a Pilot

Sometimes we wish to know the appearance of a scenario as seen by the pilot. For example if an aircraft is flying straight and level, and then executes a series of yaws, pitches, and rolls, where will the compass directions lie, and where will "up/down" be? If an aircraft is alongside but 200 m higher, where will this aircraft be seen to lie from the confines of the cockpit?

These questions are answered by rotating vectors. To determine where the compass directions are, we need to know where the aircraft is headed before it turns. These directions are only really well defined near Earth's surface, and in this case, each of them can be considered as vectors emanating from the cockpit itself, described in the aircraft's frame. For example, Fig. 3 shows that if the aircraft is flying straight and level north-east, then the north direction vector will have coordinates in the aircraft's frame of $(x, y, z) = (1/\sqrt{2}, -1/\sqrt{2}, 0)$.

Example: *This involves large distances just to ensure the method is well understood. Suppose we are flying north-east over Adelaide, pitched up at 20° at an altitude of 30,000 m. We sight an aircraft flying over Sydney at the same altitude. In which direction in the cockpit must we look to see that aircraft?*

As usual, work in the ECEF. Call the other aircraft "them", and construct a vector pointing from our location to them. This is the vector of them relative to us, or $\boldsymbol{r}_{\text{them}\leftarrow\text{us}}$. We require the components of this vector in the aircraft frame. These components are given by dotting $\boldsymbol{r}_{\text{them}\leftarrow\text{us}}$ with the three vectors describing each of the aircraft's axes. The axes are x, y, z in Fig. 3, and we'll call the vectors along each one $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$, respectively. So we need to find

$$\boldsymbol{r}_{\text{them}\leftarrow\text{us}} \cdot \boldsymbol{x}\,, \quad \boldsymbol{r}_{\text{them}\leftarrow\text{us}} \cdot \boldsymbol{y}\,, \quad \boldsymbol{r}_{\text{them}\leftarrow\text{us}} \cdot \boldsymbol{z}\,. \tag{4.11}$$

The vector $\boldsymbol{r}_{\text{them}\leftarrow\text{us}}$ is found simply by referring everything to the centre C of Earth, which implies using the ECEF:

$$\boldsymbol{r}_{\text{them}\leftarrow\text{us}} = \boldsymbol{r}_{\text{them}\leftarrow C} - \boldsymbol{r}_{\text{us}\leftarrow C}\,. \tag{4.12}$$

The vector $\boldsymbol{r}_{\text{them}\leftarrow C}$ is just their position in the ECEF, and $\boldsymbol{r}_{\text{us}\leftarrow C}$ is our own position in the ECEF. These positions are given by applying (2.2) to Adelaide's and Sydney's latitude and longitude, together with the given heights both of 30,000 m. The two cities lie at:

Adelaide:	latitude $\alpha = -34.9°$,	longitude $\omega = 138.5°$,
Sydney:	latitude $\alpha = -33.9°$,	longitude $\omega = 151.2°$.

Equation (2.2) together with the appropriate heights then gives approximately:

$$\boldsymbol{r}_{\text{them}\leftarrow C} = \begin{bmatrix} -4.67 \\ 2.57 \\ -3.55 \end{bmatrix} \times 10^6 \text{ metres}, \quad \boldsymbol{r}_{\text{us}\leftarrow C} = \begin{bmatrix} -3.94 \\ 3.49 \\ -3.65 \end{bmatrix} \times 10^6 \text{ metres}. \tag{4.13}$$

Hence

$$\boldsymbol{r}_{\text{them}\leftarrow\text{us}} = \boldsymbol{r}_{\text{them}\leftarrow C} - \boldsymbol{r}_{\text{us}\leftarrow C} = \begin{bmatrix} -7.25 \\ -9.21 \\ 0.92 \end{bmatrix} \times 10^5 \text{ metres}. \tag{4.14}$$

Now we need to find $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ at our location. First we find the local north, east, and down vectors $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$ by (4.2) or (4.3), where the heights play no part in the calculation. These were given in (4.8). These vectors must be rotated to produce our own aircraft axes. First, place our aircraft in the local NED frame flying north, straight and level. Its aircraft axes then match the local NED axes:

$$\boldsymbol{x}_0 = \boldsymbol{N}\,, \quad \boldsymbol{y}_0 = \boldsymbol{E}\,, \quad \boldsymbol{z}_0 = \boldsymbol{D}\,. \tag{4.15}$$

Next rotate the aircraft axes around $\boldsymbol{z}_0$ by 45°, and then rotate 20° around the new y-axis $\boldsymbol{y}_1$, taking care to get the angle signs right by way of the right hand rule for rotation. The sequence is:

$$\boldsymbol{x}_1 \;=\; R_{\boldsymbol{z}_0}(45°)\,\boldsymbol{x}_0$$

$$\begin{aligned}
\boldsymbol{y}_1 &= R_{\boldsymbol{z}_0}(45^\circ)\,\boldsymbol{y}_0 \\
\boldsymbol{z}_1 &= R_{\boldsymbol{z}_0}(45^\circ)\,\boldsymbol{z}_0 \qquad = \boldsymbol{z}_0 \\
\\
\boldsymbol{x} &= R_{\boldsymbol{y}_1}(20^\circ)\,\boldsymbol{x}_1 \\
\boldsymbol{y} &= R_{\boldsymbol{y}_1}(20^\circ)\,\boldsymbol{y}_1 \qquad = \boldsymbol{y}_1 \\
\boldsymbol{z} &= R_{\boldsymbol{y}_1}(20^\circ)\,\boldsymbol{z}_1\,.
\end{aligned} \tag{4.16}$$

The final vectors are approximately

$$\boldsymbol{x} = \begin{bmatrix} -0.94 \\ -0.06 \\ 0.35 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} -0.17 \\ -0.80 \\ -0.58 \end{bmatrix}, \quad \boldsymbol{z} = \begin{bmatrix} 0.31 \\ -0.60 \\ 0.74 \end{bmatrix}. \tag{4.17}$$

Dotting these with $\boldsymbol{r}_{\text{them}\leftarrow\text{us}}$ in (4.11) gives the set of components we require, so that the other aircraft turns out to have coordinates $(765, 802, 393)$ km in the frame of our aircraft. (Note that these numbers result from keeping more significant figures than are shown in the numbers above.) As a simple check, the length of this vector should be the distance of the aircraft:

$$\text{distance of aircraft} = \sqrt{765^2 + 802^2 + 393^2}\ \text{km} \simeq 1176\,\text{km}, \tag{4.18}$$

which is reasonable. To actually sight the aircraft, we first need to turn our head to our right by approximately $\tan^{-1}\frac{802}{765}$, or 46°. Again this makes sense, since the aircraft is approximately due east and we are already flying north-east. After having turned our head through 46°, we need to turn it downwards by $\tan^{-1}\frac{393}{\sqrt{765^2+802^2}}$, or about 20°.

4.3 ECEF Coordinates and Heading-Pitch-Roll Conversion

Typically, simulation software will store object locations in lat–long–height, as well as orientations in a heading–pitch–roll format: three Euler angles with respect to its local NED frame. "Heading" here means where the aircraft's x-axis points, not the direction in which it's actually travelling over the ground, which is called its track. Note that heading is not the same as yaw. Yaw is the angle between an aircraft's heading (into wind, not over ground), and the direction in which its nose points. An aircraft usually flies with no yaw, meaning that it's pointing exactly into the oncoming air flow; the air flow is approaching directly over its nose. We, together with software packages, are really assuming the aircraft is not yawed, which is a very normal way to fly. Only some extreme fliers will hold a yaw after finishing their turn.

Despite lat–long–height and heading–pitch–roll formats, an IEEE standard specifies that in order to communicate location–orientation between different machines in a distributed interactive simulation (DIS) environment, XYZ coordinates be used, along with a different set of Euler rotations. The DIS standard says that three Euler rotations are implemented by first rotating around the z-axis, then around the new y-axis, and finally around the newest x-axis. This order is useful because it corresponds to physical situations; for example, ones in which an aircraft might manoeuvre by changing its heading (about its z-axis), then pitching (about its *new* y-axis), and finally rolling (about its *newest* x-axis).

Proprietary software might well include routines to convert to and from these coordinates, but this software tends to be provided without source code and so generally is not portable. To aid in developing routines that use the DIS standard, this section describes how such conversions can be done.

The first set of DIS coordinates is the aircraft's location in the XYZ coordinates of the ECEF frame. The second set of coordinates is the set of three Euler angles that specify the aircraft's orientation by starting with its own axes coincident with the ECEF's XYZ axes, and then rotating. The three rotations are by angle ψ around $\boldsymbol{Z}$, then by θ around the rotated $\boldsymbol{Y}$, and finally by ϕ around the latest rotated $\boldsymbol{X}$.

After distribution, these numbers $(X, Y, Z, \psi, \theta, \phi)$ might need to be converted back to a lat–long–height and an orientation with respect to the local NED axes.

Converting the aircraft's location to and fro between XYZ and lat–long–height was done in (2.2)–(2.6). The next task we'll address is calculating the Euler angles ψ, θ, ϕ. As we'll see in (4.30) ahead, it's necessary to solve the problem for a general set of starting vectors, as opposed to $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}$, so we'll restate it using more general vectors. Suppose we have an orthonormal set of vectors (i.e. mutually orthogonal and all of unit length) $\boldsymbol{x}_0, \boldsymbol{y}_0, \boldsymbol{z}_0$. These are rotated by ψ about $\boldsymbol{z}_0$ to give $\boldsymbol{x}_1, \boldsymbol{y}_1, \boldsymbol{z}_1$. These new are now rotated by θ about $\boldsymbol{y}_1$ to give $\boldsymbol{x}_2, \boldsymbol{y}_2, \boldsymbol{z}_2$. Finally, these latest are rotated by ϕ about $\boldsymbol{x}_2$ to give $\boldsymbol{x}_3, \boldsymbol{y}_3, \boldsymbol{z}_3$:

$$\boldsymbol{x}_0, \boldsymbol{y}_0, \boldsymbol{z}_0 \xrightarrow[R_{\boldsymbol{z}_0}(\psi)]{} \boldsymbol{x}_1, \boldsymbol{y}_1, \boldsymbol{z}_1 \xrightarrow[R_{\boldsymbol{y}_1}(\theta)]{} \boldsymbol{x}_2, \boldsymbol{y}_2, \boldsymbol{z}_2 \xrightarrow[R_{\boldsymbol{x}_2}(\phi)]{} \boldsymbol{x}_3, \boldsymbol{y}_3, \boldsymbol{z}_3 \,. \tag{4.19}$$

The question is: given $\boldsymbol{x}_0, \boldsymbol{y}_0, \boldsymbol{z}_0$ and $\boldsymbol{x}_3, \boldsymbol{y}_3, \boldsymbol{z}_3$, what are ψ, θ, ϕ? These angles are as follows. The angle ψ is given by projecting $\boldsymbol{x}_3$ onto the $\boldsymbol{x}_0\boldsymbol{y}_0$ plane, and is the angle of this projection from $\boldsymbol{x}_0$ in the $\boldsymbol{y}_0$ direction:

$$\sin\psi = \frac{\boldsymbol{x}_3 \cdot \boldsymbol{y}_0}{\sqrt{(\boldsymbol{x}_3 \cdot \boldsymbol{x}_0)^2 + (\boldsymbol{x}_3 \cdot \boldsymbol{y}_0)^2}} \,, \quad \cos\psi = \frac{\boldsymbol{x}_3 \cdot \boldsymbol{x}_0}{\sqrt{(\boldsymbol{x}_3 \cdot \boldsymbol{x}_0)^2 + (\boldsymbol{x}_3 \cdot \boldsymbol{y}_0)^2}} \,. \tag{4.20}$$

Next, θ is the angle between $\boldsymbol{x}_3$ and the projection just mentioned:

$$\sin\theta = -\boldsymbol{x}_3 \cdot \boldsymbol{z}_0 \,, \quad \cos\theta = \sqrt{(\boldsymbol{x}_3 \cdot \boldsymbol{x}_0)^2 + (\boldsymbol{x}_3 \cdot \boldsymbol{y}_0)^2} \,. \tag{4.21}$$

Finally, ϕ is the angle from $\boldsymbol{y}_2$ to $\boldsymbol{y}_3$ in the $\boldsymbol{z}_2$ direction:

$$\sin\phi = \boldsymbol{y}_3 \cdot \boldsymbol{z}_2 \,, \quad \cos\phi = \boldsymbol{y}_3 \cdot \boldsymbol{y}_2 \,. \tag{4.22}$$

Matlab code to calculate the first two of these angles would be

```
psi   = atan2( dot(x3,y0),  dot(x3,x0) );
theta = atan2( -dot(x3,z0), sqrt( (dot(x3,x0))^2 + (dot(x3,y0))^2) );
```

For the last angle ϕ, we need $\boldsymbol{y}_2$ and $\boldsymbol{z}_2$. These are given by

$$\boldsymbol{y}_2 = \boldsymbol{y}_1 = R_{\boldsymbol{z}_0}(\psi)\boldsymbol{y}_0 \,, \quad \boldsymbol{z}_2 = R_{\boldsymbol{y}_1}(\theta)\boldsymbol{z}_1 = R_{\boldsymbol{y}_2}(\theta)\boldsymbol{z}_0 \,. \tag{4.23}$$

Appropriate Matlab code is then

```
phi = atan2( dot(y3,z2), dot(y3,y2) );
```

The sequence of steps to convert to the six DIS numbers from a lat–long–height and heading–pitch–roll is as follows:

1. Use (2.2) to convert lat–long–height (α, ω, h) to (X, Y, Z).
2. Find the local NED axes as we did in Section 4.1. That is, start with $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}$ in the ECEF frame and rotate, first through the longitude and then through the latitude, to give $-\boldsymbol{D}, \boldsymbol{E}, \boldsymbol{N}$, respectively.
3. Rotate the $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$ vectors, first by the heading around $\boldsymbol{D}$, then by the pitch around the *rotated* $\boldsymbol{E}$, and finally by the roll around the latest rotated $\boldsymbol{N}$. The resulting vectors are, respectively, the aircraft's local $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ axes (all in the ECEF frame).
4. The angles ψ, θ, ϕ are calculated from (4.20, 4.21, 4.22), where we make the identifications

$$\begin{aligned} \boldsymbol{x}_0 &= \boldsymbol{X}\,, \quad & \boldsymbol{y}_0 &= \boldsymbol{Y}\,, \quad & \boldsymbol{z}_0 &= \boldsymbol{Z}\,, \\ \boldsymbol{x}_3 &= \boldsymbol{x}\,, \quad & \boldsymbol{y}_3 &= \boldsymbol{y}\,, \quad & \boldsymbol{z}_3 &= \boldsymbol{z}\,. \end{aligned} \tag{4.24}$$

Example 1: *An aircraft is flying 10,000 m above Adelaide, heading south-east (with no wind component: the heading is taken as the direction in which the nose points), climbing at a 20° pitch, and holding a 30° roll. What are the two sets of triplets that the DIS standard requires to specify the aircraft's location and orientation?*

The location is found easily, by converting the aircraft's lat–long–height at Adelaide to XYZ using (2.2):

$$(X, Y, Z) = (-3.93, \;\; 3.48, \;\; -3.63) \times 10^6 \,\text{m}. \tag{4.25}$$

The orientation vectors $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ are found by first calculating the local NED axes as vectors in the ECEF, and then rotating these by the relevant heading, pitch and roll. Some of this calculation has been done already in Section 4.1; the local NED axis vectors are given there in (4.8). Using these, place the aircraft into the NED frame so that initially its x, y, z axes (i.e. $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ vectors) coincide with the $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$ axes respectively, and now rotate it as needed. So first set

$$\boldsymbol{x}_0 = \boldsymbol{N}\,, \quad \boldsymbol{y}_0 = \boldsymbol{E}\,, \quad \boldsymbol{z}_0 = \boldsymbol{D}\,. \tag{4.26}$$

Now rotate $\boldsymbol{x}_0, \boldsymbol{y}_0, \boldsymbol{z}_0$ each by 135° around $\boldsymbol{z}_0$ (implementing the south-east heading), calling the result $\boldsymbol{x}_1, \boldsymbol{y}_1, \boldsymbol{z}_1$. Then rotate each of these by 20° around $\boldsymbol{y}_1$ (implementing the pitch) to produce $\boldsymbol{x}_2, \boldsymbol{y}_2, \boldsymbol{z}_2$. Finally rotate each of these by 30° around $\boldsymbol{x}_2$ (implementing the roll) to produce $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$. Each of these rotations is easy to work out using the basic formula (3.7). The result is

$$\boldsymbol{x} = \begin{bmatrix} -0.366 \\ -0.564 \\ -0.741 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 0.928 \\ -0.165 \\ -0.333 \end{bmatrix}, \quad \boldsymbol{z} = \begin{bmatrix} 0.065 \\ -0.809 \\ 0.584 \end{bmatrix}. \tag{4.27}$$

Now calculate ψ, θ, ϕ from (4.20, 4.21, 4.22). To do this, set

$$\boldsymbol{x}_0 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \boldsymbol{y}_0 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \boldsymbol{z}_0 = \begin{bmatrix}0\\0\\1\end{bmatrix},$$
$$\boldsymbol{x}_3 = \boldsymbol{x}, \quad \boldsymbol{y}_3 = \boldsymbol{y}, \quad \boldsymbol{z}_3 = \boldsymbol{z}. \tag{4.28}$$

The resulting angles are

$$\psi = -123.0^\circ, \quad \theta = 47.8^\circ, \quad \phi = -29.7^\circ. \tag{4.29}$$

These, together with (X, Y, Z) given in (4.25), are the six numbers that the DIS standard uses to encode the aircraft's position and orientation.

Example 2: *Convert back from $X, Y, Z, \psi, \theta, \phi$ to lat–long–height/heading–pitch–roll.*

Again the location is found easily by applying (2.3)–(2.6). These equations return the latitude $\alpha = -34.9^\circ$, longitude $\omega = 138.5^\circ$, and height 10,000 m, as expected.

To calculate the heading, pitch, and roll, realise that these are just the ψ, θ, ϕ, respectively, that are returned by step 4 above (4.24) when we make the following identifications:

$$\boldsymbol{x}_0 = \boldsymbol{N}, \quad \boldsymbol{y}_0 = \boldsymbol{E}, \quad \boldsymbol{z}_0 = \boldsymbol{D},$$
$$\boldsymbol{x}_3 = \boldsymbol{x}, \quad \boldsymbol{y}_3 = \boldsymbol{y}, \quad \boldsymbol{z}_3 = \boldsymbol{z}. \tag{4.30}$$

So we need to calculate $\boldsymbol{N}, \boldsymbol{E}, \boldsymbol{D}$, which can be done as in Section 4.1, since we have just calculated the latitude and longitude of the aircraft. These are given in (4.8).

Next, the $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ are found by applying the three Euler rotations of ψ, θ, ϕ in the appropriate order, i.e. (4.19), to the ECEF's basis vectors $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}$. In other words, set $\boldsymbol{x}_0, \boldsymbol{y}_0, \boldsymbol{z}_0$ in (4.19) to $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}$. The three resulting vectors $\boldsymbol{x}_3, \boldsymbol{y}_3, \boldsymbol{z}_3$ will be the required $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$.

Finally, make the identifications of (4.30) to calculate ψ, θ, ϕ using (4.20, 4.21, 4.22), where the heading is ψ, the pitch is θ, and the roll is ϕ. As expected, we obtain

$$\text{heading} = 135^\circ, \quad \text{pitch} = 20^\circ, \quad \text{roll} = 30^\circ. \tag{4.31}$$

5 Concluding Remarks

All of the commonly-needed techniques in three-dimensional rotations rely on our ability to do two basic procedures:

1. Rotate a vector about any axis whatsoever, and

2. calculate components using dot products.

These calculations need to be done in just one frame. The usual one is the ECEF, which is good for all of the usual motions we associate with moving around Earth. (If we need to deal with satellites, a nonrotating frame is needed, such as one fixed to the stars.)

Rotating about arbitrary axes and taking dot products within the ECEF is all that has been done for the examples in this report.

Rotations themselves can be carried out using matrices or quaternions. Both of these are simply ways of writing down the three numbers needed to encode a rotation (an angle, and two numbers for the axis). The extra redundancy that matrices have over quaternions allows for simpler algebra when using matrices to analyse rotation on paper, but quaternions, being essentially pared-down versions of the rotation matrix, can be easier to prevent from degrading numerically inside a computer routine (but see the comment at the end of Sect. 3.4). Performance questions like this are critical in that three-dimensional models can have 10,000+ points to rotate at e.g. 30 frames per second.

When solving a problem involving three-dimensional rotations, a good approach is to break the scenario down into its building block rotations and to handle each one separately, all within the ECEF. This step-by-step approach lends itself well to being modularised in software, and is what we have used repeatedly in this report.

References

Mathematics essential to three-dimensional rotations is described in:
Lyons, L., (1998) *All You Wanted to Know About Mathematics But Were Afraid to Ask*, Cambridge University Press

Various useful calculations involving GPS coordinates are contained in:
Farrell, J., Barth, M. (1998), *The Global Positioning System and Inertial Navigation*, McGraw-Hill

Appendix A Combining Two Rotations

Earlier we said that two rotations will always combine to give a new rotation. This is of course equivalent to multiplying the two rotation matrices or quaternions. But what is the axis and angle of the new rotation? For example, suppose we rotate a vector twice:

1. First rotate it by 90° around the x-axis (call the matrix R_x and the quaternion Q_x).
2. Then rotate it by 90° around the y-axis (call the matrix R_y and the quaternion Q_y).

What is the corresponding axis and angle of the combined rotation? We do this first using matrices and then with quaternions. It will be quite evident from the answer that two rotations generally don't combine to give an obvious resulting angle and axis.

Using Matrices

The combined matrix rotation is $R_y\, R_x$:

1. $R_x : \theta = 90°, \boldsymbol{n} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$, so $R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$

2. $R_y : \theta = 90°, \boldsymbol{n} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, so $R_y = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$

3. $R_y\, R_x = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 0 \end{bmatrix}$

To find the single angle and axis of rotation that $R_y\, R_x$ is equivalent to, we could compare each element of $R_y\, R_x$ with the general expression (3.7), but this is arduous. Much easier is to use (3.7) to write down two properties of the general rotation matrix, using its trace and its transpose:

$$\begin{aligned} \operatorname{tr} R &= 1 + 2\cos\theta \\ R - R^t &= 2\sin\theta\, n^{\times} \, . \end{aligned} \tag{A1}$$

In that case, the combined rotation is around some unit vector $\boldsymbol{n}$ through angle θ, where

$$1 + 2\cos\theta = 0 \quad , \quad 2\sin\theta\, n^{\times} = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix} . \tag{A2}$$

There appear to be two solutions for $\boldsymbol{n}$ and θ here, but they both represent the same rotation, so that we can always choose the unique positive value of θ together with its corresponding $\boldsymbol{n}$. In this case:

$$\theta = 120° \quad , \quad \boldsymbol{n} = (1, 1, -1)/\sqrt{3} \, . \tag{A3}$$

Using Quaternions

The necessary quaternions are

$$\begin{aligned} Q_x &= \Big(\cos 45^\circ, \sin 45^\circ (1,0,0)\Big) = \frac{1}{\sqrt{2}}(1,1,0,0)\,, \\ Q_y &= \Big(\cos 45^\circ, \sin 45^\circ (0,1,0)\Big) = \frac{1}{\sqrt{2}}(1,0,1,0)\,. \end{aligned} \tag{A4}$$

The combined quaternion rotation is $Q_y\,Q_x$:

$$\begin{aligned} Q_y\,Q_x &= \frac{1}{2}(1,0,1,0)\,(1,1,0,0) = \frac{1}{2}(1,1,1,-1) \\ &= \left(\frac{1}{2}, \frac{\sqrt{3}}{2}\frac{(1,1,-1)}{\sqrt{3}}\right) = \left(\cos 60^\circ, \frac{(1,1,-1)}{\sqrt{3}}\sin 60^\circ\right), \end{aligned} \tag{A5}$$

which is a rotation of 120° around $(1,1,-1)/\sqrt{3}$, just as we found in (A3).

Appendix B xyz Eulers to Angle–Axis and Vice Versa

In this appendix we explain how to switch between an xyz Euler angle representation of a rotation and the angle–axis representation characterised by a single rotation matrix $R_{\boldsymbol{n}}(\theta)$ or quaternion $Q_{\boldsymbol{n}}(\theta)$. This fixed-axis Euler representation differs to the DIS standard discussed in Section 4.3 because in this appendix, the axes rotated around are fixed in space. Even so, the fixed-axis convention is also very commonly used.

xyz Eulers to Angle–Axis

Suppose a rotation is built of three Euler rotations: first by an angle α around the x-axis, then by β around y, then by γ around z. These join to give a single rotation of θ around $\boldsymbol{n}$, given by

$$R_{\boldsymbol{n}}(\theta) = R_z(\gamma)\, R_y(\beta)\, R_x(\alpha)\,, \tag{B1}$$

where the matrix $R_{\boldsymbol{n}}(\theta)$ encodes the whole rotation. If it's necessary to know $\boldsymbol{n}$ and θ explicitly, they can be found by applying (A1).

If the rotations are performed around *new* axes, then the calculation of the rotation matrix is only a little more difficult. For example, suppose the second rotation is around the new y-axis (called y'), while the third is around the newest x, called x':

$$R_{\boldsymbol{n}}(\theta) = R_{x'}(\gamma)\, R_{y'}(\beta)\, R_x(\alpha)\,. \tag{B2}$$

The first rotation is done as usual: $R_x(\alpha)$ rotates by α around $(1, 0, 0)$. The second rotation is about the new y-axis, whose vector is

$$\boldsymbol{n}_1 \equiv R_x(\alpha) \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} . \tag{B3}$$

The last rotation, by γ, needs to be done around the new x-axis. This axis is represented by the vector

$$\boldsymbol{n}_2 \equiv R_{\boldsymbol{n}_1}(\beta) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} , \tag{B4}$$

so that the final rotation is

$$R_{\boldsymbol{n}}(\theta) = R_{\boldsymbol{n}_2}(\gamma)\, R_{\boldsymbol{n}_1}(\beta)\, R_x(\alpha)\,. \tag{B5}$$

These calculations can of course also be done using quaternions, as detailed in Appendix A.

xyz Angle–Axis to Eulers

If the xyz rotation order of (B1) is used, a single rotation matrix R can be converted to three rotations using the Euler angles α, β, γ, where s can be chosen as either $+1$ or -1:

$$\begin{aligned}\sin\alpha &= \frac{sR_{32}}{\sqrt{1-R_{31}^2}}\,, \quad \cos\alpha = \frac{sR_{33}}{\sqrt{1-R_{31}^2}}\\ \sin\beta &= -R_{31}\,, \qquad \cos\beta = s\sqrt{1-R_{31}^2}\\ \sin\gamma &= \frac{sR_{21}}{\sqrt{1-R_{31}^2}}\,, \quad \cos\gamma = \frac{sR_{11}}{\sqrt{1-R_{31}^2}}\,. \end{aligned} \tag{B6}$$

The value of s is immaterial in the sense that either choice will give three angles that have the same effect as the original rotation. Care is needed to avoid oversimplifying these equations: we must remember that (B6) does not imply that e.g. $\alpha = \tan^{-1}(R_{32}/R_{33})$, since the $\tan^{-1}$ function only gives angles in the range $-90°\rightarrow +90°$. Nor does it imply that β must equal $-\sin^{-1} R_{31}$, because of a similar restriction on the $\sin^{-1}$ function.

These equations can be implemented in Matlab using the code:

```
alpha = atan2(s*R(3,2), s*R(3,3));
beta  = atan2(-R(3,1),  s*sqrt(1-R(3,1)^2));
gamma = atan2(s*R(2,1), s*R(1,1));
```

Appendix C Confusing Euler Angle Orientation with Incremental Rotation

As we saw in Sect. 3.2, an object's orientation can be specified by giving three Euler angles: angles through which the object can be turned in some preset order from a base position, which will result in the required orientation. This is useful and unambiguous, but it can lead to some confusion when implemented in an overly rigid way.

The problem can be shown as follows. Suppose we have written computer code that draws some object, and supplies three graphical sliders that allow us to specify Euler angles to change its orientation. We can alter any of the angles at any time, and the software will read the current values of the three angles, interpret them as Euler angles, and use them to rotate a preset base orientation of the object, first around the x-axis, then around y, and finally around z. The result is then displayed on the computer screen.

When we start the programme, this graphical interface is always drawn with the three sliders each set to zero, and the object in its base position. If we alter the x-slider to read 10° and leave the other two sliders at zero, then the three angles are read by the software, and a rotation of $R_z(0)\,R_y(0)\,R_x(10°)$ is applied. The object duly rotates around x by 10°. If we increase the x-slider to 15°, the software reads all the sliders again and applies a rotation of $R_z(0)\,R_y(0)\,R_x(15°)$ to the *base* orientation. The effect is that the object is seen to rotate around x through a further 5°. We then set the x-slider back to zero, the three angles are again read, and the result is that the base orientation is rotated by $R_z(0)\,R_y(0)\,R_x(0)$, or nothing at all. So changing the x-slider by itself rotates the object about the x-axis, which is perfectly reasonable and expected.

Similarly if we change only the y-slider, or only the z-slider, the same sort of intuitive effects happen. Setting the y-slider to 10° with the others at zero means that the software reads the three angles and applies a rotation of $R_z(0)\,R_y(10°)\,R_x(0)$ to the base orientation, so that the object rotates around y by 10°, and similarly it will revert to the base position if we set the y-slider back to zero.

Suppose we now change two of the angles; we'll ignore the z-slider as it's not needed to make the following point. Start with all sliders at zero so that the object is drawn in the base orientation. Then set the x-slider to 10°. The rotation is $R_y(0)\,R_x(10°)$, so the object rotates around x by 10°. Now set the y-slider to 10°. The new rotation is $R_y(10°)\,R_x(10°)$ from the *base* orientation, with the result that the object is first rotated around x by 10° and then around y by 10°. So far, the behaviour matches our intuition.

Now increase the x-slider to read 15°. The rotation is now $R_y(10°)\,R_x(15°)$ from the base orientation, so that the object is first rotated around x by 15° and then around y by 10°. The procedure is shown as follows:

x-slider setting	y-slider setting	Resulting rotation of base orientation
0	0	$R_y(0)\ R_x(0)$
10°	0	$R_y(0)\ R_x(10°)$
10°	10°	$R_y(10°)\ R_x(10°)$
15°	10°	$R_y(10°)\ R_x(15°)$

But the effect of this latest rotation is not what we might have expected. It is reasonable to believe, based on trying each slider in turn on its own, that each slider rotates about its respective axis. When we increased the x-slider from 10° to 15°, most users would expect the object to rotate by a further 5° about x. But this is not what happens. The software does what it was designed to do: it rotates the base orientation using

$$R_y(10^\circ)\ R_x(15^\circ)\,. \tag{C1}$$

The rotation suggested by our intuition is an *increment* of 5° about x on the last orientation—which was represented by $R_y(10^\circ)\ R_x(10^\circ)$ acting on the base orientation—or

$$R_x(5^\circ)\ R_y(10^\circ)\ R_x(10^\circ)\,. \tag{C2}$$

Expressions (C1) and (C2) are not the same. In fact (C1) could have been written

$$R_y(10^\circ)\ R_x(5^\circ)\ R_x(10^\circ)\,, \tag{C3}$$

which differs from (C2) in that two rotations are swapped. Because rotation is noncommutative, these two different matrices (C2) and (C3) produce different orientations when multiplying the base orientation. To reiterate, our intuition expected that by making three slider adjustments, first incrementing the x-slider (from zero) by 10°, then incrementing the y-slider (from zero) by 10°, then incrementing the x-slider by 5°, that (C2) would result. In fact, (C1) or equivalently (C3) resulted. That might be unintuitive, but the mathematics and the computer software have done exactly what was asked of them.

Our confusion over the effects of the sliders seems to reach its worst if the y-slider is first set to rotate through −90°, and then the x- and z-sliders are changed arbitrarily. Because the following identity holds for all angles α, β,

$$R_z(\alpha)\ R_y(-90^\circ)\ R_x(\beta) = \begin{bmatrix} 0 & -\sin(\alpha+\beta) & -\cos(\alpha+\beta) \\ 0 & \cos(\alpha+\beta) & -\sin(\alpha+\beta) \\ 1 & 0 & 0 \end{bmatrix}\,, \tag{C4}$$

the total rotation—being a function only of the sum of the x- and z-slider angles—does not distinguish between the x- and z-sliders. Thus the x- and z-sliders have the same effect, making it appear that we have lost one rotation axis somewhere, as if, to use an oft-quoted phrase, "the x-axis has been rotated onto the z-axis" (!) But axes are not set on hinges; there is no such thing as a flexible axis, and axes cannot be rotated onto each other. The software is doing exactly what it was designed to do. Whether our intuition would prefer a different behaviour is another matter entirely.

This apparent loss of one axis is erroneously labelled gimbal lock by some graphics programmers, who confuse it with the real numerical Euler angle problem described earlier on page 7. But there is no such problem with Euler angles in software rotations such as described above, and this use of the term is a misnomer. Unfortunately, historically these misinterpretations went hand in hand with using matrices to perform rotations, with the result that matrices and Euler angles are sometimes seen as not up to the task. This is definitely wrong; both matrices and Euler angles can handle any rotation asked of them.

Restoring an Intuitive Feel to the Sliders If we wish the sliders to act in an intuitive way, then each time we adjust any slider, we must not have the software read the three slider values and rotate the base orientation through them all in some preset order. Rather, if we adjust the x-slider by 2°, the software should read the values, note what has changed, and rotate the *current* orientation about the x-axis through the increment of 2°. Not only does this make the sliders and rotations behave as our intuition suggests, but it also makes for faster processing, since only one rotation is ever performed per slider interaction, instead of three.

Appendix D Quaternions Used in Computing: SLERP

Rotation theory is important when creating a smooth fly through of a scene in which certain way points have been specified along with camera orientations at those points. A camera's orientation can be specified by a rotation matrix or quaternion that generates that orientation by rotating the initial orientation. So the question arises as to how best to interpolate the matrices or quaternions, in order to generate rotations that act on the initial orientation to give visually acceptable intermediate orientations.

For example, if two way-point orientations specified by rotation matrices R_0 and R_1 are specified, then we might suppose that linear combinations of these with appropriate weightings will suffice to rotate the initial orientation to create intermediate orientations. But such is not the case; the intermediate rotation matrices are not simply given by linear interpolations between R_0 and R_1. For example in the simpler case of rotations in the xy plane, if the rotation matrix for 30° is $R_z(30°)$ using (3.4), and that for 40° is $R_z(40°)$, then the matrix

$$0.9\, R_z(30°) + 0.1\, R_z(40°) \tag{D1}$$

is *not* equal to $R_z(31°)$; being not orthogonal, it's not a rotation matrix at all. Even when orthogonalised using (3.13), what results still does not equal $R_z(31°)$. So a more sophisticated matrix interpolation is needed.

Whatever results are applicable to matrices, quaternions seem to be easier to use; and certainly most of the research into interpolating orientations seems to revolve around quaternions. This seems to be mostly due to the fact that because of its numerically intensive nature, rotation research tends to be mainly the domain of computer animation programmers; but unfortunately once various quaternion routines had been worked out and coded, the code got copied, understood, and used extensively; and quaternions became the de facto way to solve problems of this sort. Nonetheless, as we'll show in this appendix, quaternions are very easily interpolated to give more acceptable interpolation results than are traditionally obtained using matrices.

The fact that quaternions have a unit length given by a simple sum of squares of their components means they can be treated as vectors in Euclidean four dimensions, so are able to be visualised as points on the surface of a "sphere", as shown in Figure D1. In this figure, the initial orientation is represented by the quaternion $(1, 0, 0, 0)$, because this is the identity quaternion that doesn't rotate at all, shown by applying (3.17):

$$(0, \boldsymbol{x}') = (1, 0, 0, 0)\, (0, \boldsymbol{x})\, (1, 0, 0, 0) = (0, \boldsymbol{x})\,. \tag{D2}$$

We wish to generate a whole series of quaternions, each of which multiplies the initial orientation to produce some intermediate one. We must pass through the way-point quaternion $(0.9, 0.1, 0.1, 0.4)$ and finish at the final quaternion $(0.7, 0.6, 0.2, 0.3)$. The very simplest way is to make a linear interpolation, in angle, between way points on the sphere's surface, and so is called *spherical linear interpolation*, or SLERP. To see how SLERP is used to interpolate between two quaternions, draw them as vectors $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ lying in a plane in Figure D2. (Previously we used e.g. Q to denote a quaternion. Here we use bold face vector notation to reinforce our treatment of quaternions as vectors, since that's what

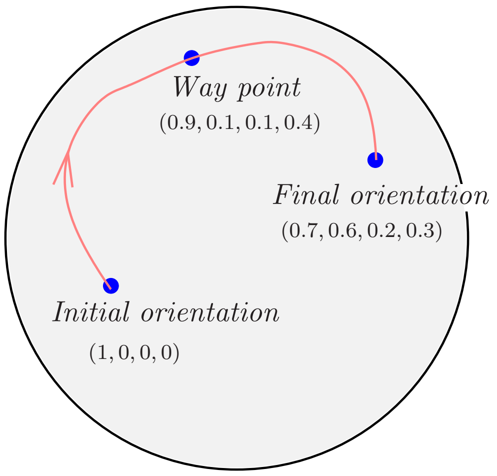

Figure D1: Schematic of a four-dimensional sphere, each surface point of which represents a quaternion. Any orientation of some system is given by a quaternion that rotates the initial orientation. Given way-point orientations, the task is to find a path connecting them that then picks out the set of interpolating quaternions that each generate an acceptable intermediate orientation of the system

picturing them as points on the surface of a four-dimensional sphere allows us to do.) The angle θ between them is given by the usual dot product:

$$\cos\theta = \frac{\boldsymbol{q}_1 \cdot \boldsymbol{q}_2}{|\boldsymbol{q}_1|\,|\boldsymbol{q}_2|} = \boldsymbol{q}_1 \cdot \boldsymbol{q}_2\,, \tag{D3}$$

where the dot product is the usual sum of pairwise products of components. Use a parameter t running from zero to one, values of which generate intermediate quaternions such as the $\boldsymbol{q}$ shown in Figure D2. Straightforward two-dimensional vector analysis gives

$$\boldsymbol{q} = \frac{\sin{(1-t)\theta}}{\sin\theta}\,\boldsymbol{q}_1 + \frac{\sin t\theta}{\sin\theta}\,\boldsymbol{q}_2\,. \tag{D4}$$

This is the standard SLERP formula for generating intermediate quaternions. In Figure D1 we first interpolate for various values of t between the initial orientation and the way point, and then do the same for the way point and the final orientation. Remember that each quaternion generated does *not* multiply the orientation just produced; rather, it multiplies the initial orientation.

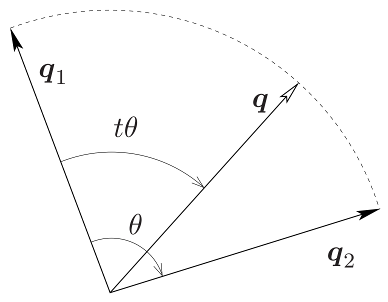

Figure D2: The quaternion $\boldsymbol{q}$ is interpolated between quaternions $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$

SLERP is popular among graphics programmers, but there is some recognition that this is partly by default because of the way routines get shared around. SLERP is not the only quaternion tool used, and it has even been argued as generally not desirable to use, partly because of its high processing demands, and partly because other algorithms (such as *normalised quaternion linear interpolation*) bring other benefits to the programmer in terms of the "look and feel" of the resulting rotations. Nevertheless, the above description of SLERP shows the general idea of the current approach to interpolation within the computer graphics community.

DISTRIBUTION LIST

Using Rotations to Build Aerospace Coordinate Systems

Don Koks

	Number of Copies
DEFENCE ORGANISATION	
Task Sponsor	
Director General Aerospace Development	1
S&T Programme	
Chief Defence Scientist	1
FAS Science Policy	
AS Science Corporate Management	
Director General Science Policy Development	
Counsellor, Defence Science, London	Doc. Data Sheet
Counsellor, Defence Science, Washington	Doc. Data Sheet
Scientific Adviser to MRDC, Thailand	Doc. Data Sheet
Scientific Adviser Joint	1
Navy Scientific Adviser	Doc. Data Sheet and Dist. List
Scientific Adviser, Army	Doc. Data Sheet and Dist. List
Air Force Scientific Adviser	1
Scientific Adviser to the DMO	Doc. Data Sheet and Dist. List
Platform Sciences Laboratory	
Director of PSL (Corporate Leader Air)	Doc. Data Sheet and Exec. Summary
Systems Sciences Laboratory	
EWSTIS (soft copy for accession to web site)	1 (pdf format)
Chief, Electronic Warfare and Radar Division	Doc. Data Sheet and Dist. List
Research Leader, EWS Branch, EWRD	1
Research Leader, MR Branch, EWRD	1
Head, AS group, EWRD	1
Head, SLEW group, EWRD	1
Head, MS group, EWRD	1
Don Koks, EWRD	5
Shane Kelly, EWRD	1
Mark Petrusma, WSD	1

Robert Anderson, WSD	1
David Marlow, AOD, Fishermans Bend	1
DSTO Library and Archives	
Library, Fishermans Bend	Doc. Data Sheet
Library, Edinburgh	2
Library, Sydney	Doc. Data Sheet
Library, Stirling	Doc. Data Sheet
Library, Canberra	Doc. Data Sheet
Defence Archives	1
Capability Development Group	
Director General Maritime Development	Doc. Data Sheet
Director General Integrated Capability Development	1
Director General Capability and Plans	Doc. Data Sheet
Assistant Secretary Investment Analysis	Doc. Data Sheet
Director Capability Plans and Programming	Doc. Data Sheet
Director General Australian Defence Simulation Office	Doc. Data Sheet
Chief Information Officer Group	
Fltlt Barry Murry, R1-3-A100, Chief Information Office, Russell Offices, Russell Drive, Canberra ACT 2600	1
Director General Information Policy and Plans	Doc. Data Sheet
AS Information Strategy and Futures	Doc. Data Sheet
AS Information Architecture and Management	Doc. Data Sheet
Director General Information Services	Doc. Data Sheet
Strategy Group	
Director General Military Strategy	Doc. Data Sheet
Assistant Secretary Strategic Policy	Doc. Data Sheet
Assistant Secretary Governance and Counter-Proliferation	Doc. Data Sheet
Navy	
Maritime Operational Analysis Centre, Building 89/90, Garden Island, Sydney	
Deputy Director (Operations)	Doc. Data Sheet and Dist. List
Deputy Director (Analysis)	Doc. Data Sheet and Dist. List
Director General Navy Capability, Performance and Plans, Navy Headquarters	Doc. Data Sheet
Director General Navy Strategic Policy and Futures, Navy Headquarters	Doc. Data Sheet

Air Force

SO (Science), Headquarters Air Combat Group, RAAF Base, Williamtown NSW 2314	Doc. Data Sheet and Exec. Summary

Army

ABCA National Standardisation Officer, Land Warfare Development Sector, Puckapunyal	Doc. Data Sheet (pdf format)
SO (Science), Land Headquarters (LHQ), Victoria Barracks, NSW	Doc. Data Sheet and Exec. Summary
SO (Science), Deployable Joint Force Headquarters (DJFHQ)(L), Enoggera QLD	Doc. Data Sheet

Joint Operations Command

Director General Joint Operations	Doc. Data Sheet
Chief of Staff Headquarters Joint Operation Command	Doc. Data Sheet
Commandant, ADF Warfare Centre	Doc. Data Sheet
Director General Strategic Logistics	Doc. Data Sheet
COS Australian Defence College	Doc. Data Sheet

Intelligence and Security Group

DGSTA, DIO	1
Manager, Information Centre, DIO	1 (pdf format)
Assistant Secretary Capability Provisioning, DSD	Doc. Data Sheet
Assistant Secretary Capability and Systems, DIGO	Doc. Data Sheet

Defence Materiel Organisation

Deputy CEO	Doc. Data Sheet
Head Aerospace Systems Division	Doc. Data Sheet
Head Maritime Systems Division	Doc. Data Sheet
Head Electronic and Weapon Systems Division	Doc. Data Sheet
Programme Manager Air Warfare Destroyer	Doc. Data Sheet

Defence Libraries

Library Manager, DLS-Canberra	Doc. Data Sheet

OTHER ORGANISATIONS

National Library of Australia	1
NASA (Canberra)	1
State Library of South Australia	1

UNIVERSITIES AND COLLEGES

Australian Defence Force Academy Library	1

Head of Aerospace and Mechanical Engineering, ADFA	1
Serials Section (M List), Deakin University Library, Geelong, Victoria	Doc. Data Sheet
Hargrave Library, Monash University	Doc. Data Sheet
Librarian, Flinders University	1

OUTSIDE AUSTRALIA

INTERNATIONAL DEFENCE INFORMATION CENTRES

US: Defense Technical Information Center	1 (pdf format)
UK: DSTL Knowledge Services	1 (pdf format)
Canada: Defence Research Directorate R&D Knowledge and Information Management (DRDKIM)	1
NZ: Defence Information Centre	1

ABSTRACTING AND INFORMATION ORGANISATIONS

Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
Materials Information, Cambridge Scientific Abstracts, US	1
Documents Librarian, The Center for Research Libraries, US	1

INFORMATION EXCHANGE AGREEMENT PARTNERS

National Aerospace Laboratory, Japan	1
National Aerospace Laboratory, Netherlands	1

SPARES

DSTO Edinburgh Library	5

Total number of printed copies: 43

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA	1. CAVEAT/PRIVACY MARKING
2. TITLE Using Rotations to Build Aerospace Coordinate Systems	3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)
4. AUTHOR Don Koks	5. CORPORATE AUTHOR Systems Sciences Laboratory P.O. Box 1500 Edinburgh, SA 5111 Australia

6a. DSTO NUMBER DSTO–TN–0640	6b. AR NUMBER AR–013–424	6c. TYPE OF REPORT Technical Note	7. DOCUMENT DATE August, 2008

8. FILE NUMBER 2004/1092940	9. TASK NUMBER AIR 00/069	10. SPONSOR DGAD	11. NO. OF PAGES 31	12. NO. OF REFS 2

13. URL OF ELECTRONIC VERSION http://www.dsto.defence.gov.au/corporate/reports/DSTO–TN–0640.pdf	14. RELEASE AUTHORITY Chief, Electronic Warfare and Radar Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved For Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, P.O. BOX 1500, EDINBURGH, SA 5111, AUSTRALIA

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DEFTEST DESCRIPTORS

Rotation	Axes of rotation
Coordinates	Space navigation

19. ABSTRACT

Presented here are the main techniques necessary to understand rotations in three dimensions, for use with global visualisation and aerospace simulations. Relevant techniques can be extremely difficult to find in textbooks, so some useful examples are collected here to highlight these techniques.
The three standard aerospace coordinate systems are described and built using rotations. The mathematics of rotations is described, using both matrices and quaternions. The necessary calculations are given for analysing standard scenarios that involve the Global Positioning Satellite system for finding line-of-sight directions on Earth, as well as for visualising the world from a cockpit, and for converting to and from the standard software protocol for distributed interactive simulation environments.
Appendices then discuss combining rotations, conversions with a particular type of Euler angle convention, the dangers of confusing Euler angles with incremental rotations for software writers, and finally there is a short dicussion of interpolation of rotations in computing.